Dynamic Fractional Resource Scheduling for Cluster Platforms

Mark Stillwell (4th year PhD student)
Department of Information & Computer Sciences
University of Hawai'i at Mānoa, Honolulu, U.S.A.
Advisor: Henri Casanova

***Abstract*—We propose a novel approach, called Dynamic Fractional Resource Scheduling (DFRS), to share homogeneous cluster computing platforms among competing jobs. DFRS leverages virtual machine technology to share node resources in a precise and controlled manner. A key feature of this approach is that it defines and optimizes a user-centric metric of performance and fairness. We explain the principles behind DFRS and its advantages over the current state of the art, develop a model of resource sharing, and summarize results from two different simulation experiments: one comparing various heuristics in an off-line setting and another comparing our heuristics to current technology in an on-line setting. Finally, we summarize our conclusions and describe our plans for future research.**

I. Introduction

While clusters provide an improvement in cost over traditional supercomputers, they still represent significant capital investments. Thus, clusters are typically expected to run multiple instances of user applications, or *jobs*, which have specific resource requirements. The assignment of jobs to nodes in space and/or time in such a way that these various requirements can be met is called *job scheduling*.

We propose a novel approach to job scheduling called *Dynamic Fractional Resource Scheduling*, or DFRS, that makes use of virtual machine technology to share resources. Our approach is to attempt to optimize a well-defined performance metric at runtime. We contend that algorithms that perform well by this metric will also perform well by other traditional metrics, and in particular will outperform current job scheduling approaches.

II. Current Approaches

Since most clusters have traditionally been used for scientific and high performance computing (HPC) the existing literature tends to look at the problem from this perspective. In the past few years there has also been an interest in using clusters for service hosting and data processing applications. We seek to define an approach that is compatible with both of these types of workload.

In scientific workloads jobs consist of one or more parallel *tasks*, which must each run on individual cluster nodes. Tasks within a job may communicate with each other, but generally do not require user interaction. Most jobs are expected to complete and exit after some finite amount of time.

The most commonly used cluster sharing technique for these types of workloads is *batch scheduling*. In this paradigm users create requests to run programs using a number of nodes for a given amount of time, and these requests are then put into queues. At some future time a request is used to create a job that is then given exclusive access to a set of nodes. Requests are generally handled on a first-come-first-served basis. An additional *backfilling* mechanism may be used to select smaller/shorter tasks out of turn in order to improve average response time and minimize the number or idle nodes. Unfortunately, the user runtime estimates required for backfilling are highly unreliable [1]. Also, the node-level exclusive allocations used lead to poor utilization of nodes and node resources.

Gang scheduling is a paradigm in which the entire parallel machine is multiplexed via distributed context switching so that tasks within the same job are run during the same time slices. One problem with gang scheduling is that it requires *coordinated* distributed context switching. This is high-overhead, which leads to long time slices and thus severely limits the expected benefit in terms of resource utilization. Another problem with gang scheduling is that the original models did not take into account the memory requirements of running jobs when packing them into time slots [2], which leads to expensive swaps to disk. In spite of recent developments and improvements, gang scheduling is not generally used in production environments.

In contrast to the HPC context, jobs in service hosting environment tend to be long-lived and require little inter-node communication. The primary goals here are to ensure performance isolation and meet service level agreements (SLAs) [3].

We frame the scheduling problem for service hosting as an optimization problem with a well defined objective function correlated to other metrics of performance. A similar approach has been described in the literature [4], but our work possesses two key advantages. First we seek algorithms that can account for an arbitrary number of resource dimensions instead of only one or two.

Second, we define and optimize a universally applicable metric that captures notions of performance and fairness across different application domains. As a result our approach is applicable to both service hosting and HPC environments.

III. Virtual Machine Technology

Virtualization makes it possible to pool and redistribute multiple discrete time-shared resources in a fluid manner. For example, the Xen Credit CPU scheduler can allow 3 serial jobs to each receive 1/3 the total CPU capacity of a dual-core system [5]. Further, studies have shown that VM monitors can enforce constraints on resource utilization accurately and reliably in the presence of competing processes [6]. Virtualization also allows for preemption and the migration of processes between nodes. Migration is beneficial for gang scheduling [7], and the same should hold for time-sharing based on virtualization.

IV. Problem Framework

Jobs consist of one or more *tasks*, each of which is a more or less independent instance of the program. It is reasonable to assume that tasks in a job have similar resource requirements and may need to communicate with each other. We propose to run these tasks within independent virtual machines whose resource consumption levels can be carefully controlled.

Resources, such as memory or disk space, that cannot be time-shared are referred to as *fixed*, while those, such as CPU, network, or disk I/O bandwidth, that can are referred to as *fluid*. For each fixed resource a task has a *requirement* in that resource equal to a certain fraction of what is available on a node. For each fluid resource a task has a *need* in that resource equal to the fraction of the time the task would keep that resource busy if it were running alone on the node. The total fixed requirements of the tasks on a given node can never exceed 100% of any resource. When the total fluid needs of the tasks on a node exceed 100% of some resource, the performance of some of the tasks must be degraded.

We make the strong assumption that within a task the actual utilization levels of all fluid resources are linearly correlated, so that allocating a restricted amount of one resource equal to the need multiplied by a given factor $\alpha \in [0, 1]$ will result in all of the task's fluid resource utilization levels being equal to the need in each resource multiplied by α. We further assume that the overall performance of the job is also multiplied by α. These assumptions are easily justifiable for a number of different application classes, such as web servers or parallel computations. We refer to α as the *yield*.

V. The Off-line Problem

In the off-line allocation problem for continuous serial jobs with unchanging resource requirements we consider a set of nodes N and a set of jobs J. Each Job j is denoted by a pair of vectors $(\vec{s_j}, \vec{d_j})$ such that $\vec{s_j}$ represents fixed resource requirements and $\vec{d_j}$ the fluid resource needs. A solution is a number $\alpha \in [0, 1]$, representing the minimum yield, and a mapping $f : J \rightarrow N$ and a valid solution is a pair that does not require the allocation of more than 100% of any resource on any node.

The goal of the optimization problem is to find a valid solution that maximizes the minimum *yield*. This problem definition builds on our previous work [8], and we have proved that the problem is strongly NP-hard [9].

A. Results

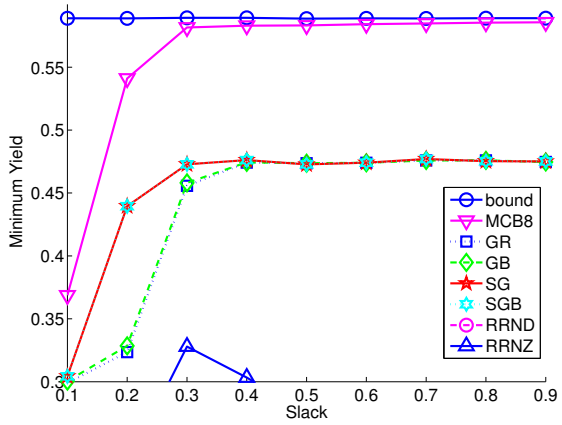
In our previous work we considered the static resource sharing problem for CPU-bound jobs with memory requirements [8], [9]. This corresponds to the off-line problem with a single fluid resource need and a single fixed resource requirement. We gave a mixed integer linear program (MILP) formulation through which the optimal solution can be computed (only practical for small instances), proposed a number of simple algorithms and heuristics, and then performed simulation experiments to evaluate the relative performance of those heuristics.

We conducted these simulations on synthetic problem instances defined based on the number of nodes, the number of jobs, and the total amount of free memory, or *memory slack*, in the system. In general (but not always) the greater the slack the greater the number of feasible solutions. Experimental details are provided in our previous work [8], [9].

The heuristics we considered fall into three groups: 1) greedy heuristics that allocate jobs to nodes with the least computational load (GR, GB, SG, and SGB), 2) randomized rounding heuristics that attempt to use the relaxation of the MILP to an LP as a first step toward finding a solution (RRND and RRNZ), and 3) heuristics that apply a multi-capacity bin-packing heuristic at each step of a binary search, seeking to maximize the minimum yield (MCB8).

Figure 1 shows minimum yield achieved vs. slack. The algorithm based on the multi-capacity bin-packing heuristic (MCB8) is best. In fact, it outperforms or equals all other algorithms nearly across the board in terms of minimum yield and failure rate, while exhibiting relatively low run times. The sorted greedy algorithms (SG or SGB) lead to reasonable results and could be used for instances with very large numbers of tasks. A more detailed explanation of the experimental results can be found in our published work [8], [9].

Figure 1. Minimum Yield vs. Slack for Large Instances



VI. The On-line Problem

Consider a set of nodes, N, that must service a stream of user jobs J. Each job j has a release time, r_j, before which the scheduler cannot start the job and has no information about it, a number of tasks, T_j, fixed resource requirement and fluid resource need vectors $\vec{s_j}$ and $\vec{d_j}$ as described in the formulation of the off-line problem, and an amount of work to be done, w_j, that is not known until the task completes. We take w_j to be the runtime of job j on a dedicated system. The scheduler can assign job tasks to nodes, migrate tasks between nodes, or set job yield values subject to the constraints described previously. Preempting jobs and migrating some (or all) of the tasks of a job to a different set of nodes have costs, both in terms of bandwidth consumed and in terms of penalizing the performance of the selected job. For each job j the value $y_j(t)$ represents the yield of the job at time t. When a job is not running the yield is said to be zero. The *virtual time* of a job at time t is the total subjective time experienced by the job at time t, and is equal to $\int_{r_j}^{t} y_j(\tau)d\tau$. A job completes when its virtual time equals or (in the case of discrete time units) exceeds w_j.

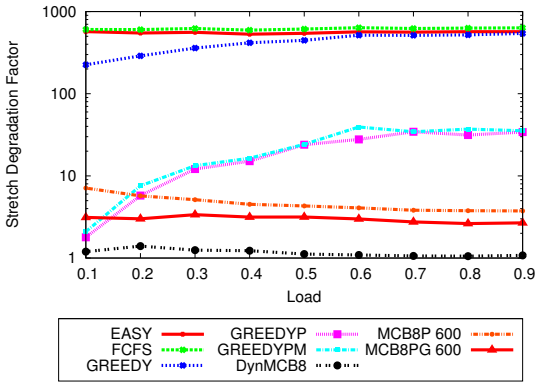
In this formulation of the problem, our goal is to minimize the maximum *stretch* (also called the *slowdown*, the ratio of the time the job is in the system to its run time on a dedicated system, a well-established metric [2], [10]), but without making any assumptions of knowledge of job run times as in most traditional approaches. We propose to approach the problem as a sequence of instances of the off-line problem and contend that this will lead to lower stretch values than current techniques.

A. Results

We used a discrete event simulator to evaluate the performance of our algorithms relative to the baseline FCFS and EASY algorithms used in production batch schedulers. The DFRS algorithms considered can be broken down into two groups: those based on the greedy heuristics for solving the off-line problem and those based on the multi-capacity bin-packing heuristics. In addition to the basic greedy heuristic (Greedy) we also considered a version that uses preemption to pause older jobs in order to run newer jobs (GreedyP) and a version that also possesses the capability to greedily migrate tasks (GreedyPM). The DynMCB8 algorithm aggressively applies the MCB8 algorithm on every event, while MCB8P algorithm applies MCB8 periodically (every 600 seconds in our experiments) regardless of events. The MCB8PG algorithm combines these approaches by attempting to greedily schedule new jobs, and then applying the MCB8 algorithm periodically.

For the experimental workloads we used synthetic traces based on a well-known model [11] and augmented with CPU needs and memory requirements [12]. To study the impact of load, we scaled the job inter-arrival times in each generated trace to create 9 new sets of traces with identical job mixes but offered load [10] levels of 0.1 to 0.9 in increments of 0.1.

Figure 2. Maximum Stretch Degradation vs. Offered Load



For a given instance, and for each algorithm, we compute the maximum stretch *degradation factor*. A value of 1 means that the algorithm is the best for that instance, while a value of 10 means that there was an algorithm in that experiment that achieved a maximum stretch 10 times lower.

Figure 2 plots average degradation factors vs. load for the algorithms when applied to scaled synthetic workloads, using a logarithmic scale on the y-axis. All of the DFRS approaches perform as well or better than traditional approaches in the average case, though the purely greedy algorithms can have worse worst case performance. The MCB heuristics perform better than the greedy algorithms overall, though GreedyP and GreedyPM do better than the MCB periodic algorithms for low load conditions. The clear overall winner for

this case is DynMCB8. However, additional experiments show that this approach can cause large numbers of migrations, and when the performance impact of these migrations is taken into account it suffers considerably [12]. A complete treatment that considers migration costs and additional experiments using traces based on a real workload is included in our most recent paper [12].

VII. Conclusion

In this summary of our research we have explained the basic principles of Dynamic Fractional Resource Scheduling (DFRS), a novel approach for allocating cluster resources among competing jobs that relies on virtual machine technology. We have given a formalization of the off-line problem and through simulation experiments have identified a heuristic that runs quickly and provides solutions on par with or superior to those of its competitors. We have further described the on-line version of this problem and demonstrated, again through simulation experiments, that DFRS has the potential to provide a tremendous improvement in performance over traditional approaches.

This work is currently being extended in several directions. While we have primarily studied CPU-bound jobs with fixed memory requirements, our framework is also capable of considering situations where jobs have resource requirements in multiple additional dimensions, such as network bandwidth, disk space, or even software licensing restrictions. The approach is also compatible with setting user-level priorities or minimum resource shares in order to satisfy service level agreements, and we have additional research examining these issues that is currently under consideration for publication.

An important consideration of our approach is the cost associated with adapting to changes to the set of currently active jobs. In our on-line scenario after a mapping of jobs to nodes has been computed and instantiated existing jobs will eventually complete and new jobs may be submitted. Additionally, it is possible, even likely, for job resource needs to change over time. Our current approach is to compute a new resource allocation and migrate jobs if necessary, however in some cases this can lead to a large number of migrations. The adaptation version of our problem bounds the amount of migration activity allowed on each event, creating a trade-off between initial schedule quality and migration costs.

It is our goal to create and evaluate a practical implementation of our ideas. This implementation is in development as a plug-in to an existing virtual machine manager and we expect to have a working prototype in the near future. We are currently using the already developed portions of the system to study algorithms for the on-line discovery of virtual machine resource needs.

References

[1] C. B. Lee and A. E. Snavely, "On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions," *Intl. J. of High Performance Computing Applications*, vol. 20, no. 4, pp. 495–506, 2006.

[2] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling – a status report," in *Proc. of the 10th Intl. Workshop on Job Scheduling Strategies for Parallel Processing*, ser. LNCS, 2004, vol. 3277, pp. 1–16.

[3] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat, "Model-based resource provisioning in a web service utility," in *Proc. of the 4th USENIX Symp. on Internet Tech. and Systems*, 2003, pp. 57–71.

[4] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *Proc. of the 16th Intl. Conf. on the World Wide Web*, 2007, pp. 331–340.

[5] D. Gupta, L. Cherkasova, and A. M. Vahdat, "Comparison of the three CPU schedulers in Xen," *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, 2007.

[6] D. Schanzenbach and H. Casanova, "Accuracy and responsiveness of CPU sharing using Xen's cap values," University of Hawai'i at Mānoa Department of Information and Computer Sciences, Tech. Rep. ICS2008-05-01, May 2008. [Online]. Available: http://www.ics.hawaii.edu/research/tech-reports/ICS2008-05-01.pdf

[7] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam, "An integrated approach to parallel scheduling using gang-scheduling, backfilling and migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 236–247, 2003.

[8] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation using virtual clusters," in *Proc. of the 9th IEEE Intl. Symp. on Cluster Computing and the Grid*, 2009, pp. 260–267.

[9] ——, "Resource allocation using virtual clusters," University of Hawai'i at Mānoa Department of Information and Computer Sciences, Tech. Rep. ICS2008-09-01, Sep. 2008. [Online]. Available: http://www.ics.hawaii.edu/research/tech-reports/ics2008-09-15.pdf

[10] A. Batat and D. G. Feitelson, "Gang scheduling with memory considerations," in *Proc. of the 14th Intl. Parallel and Distributed Processing Symp.*, 2000, pp. 109–114.

[11] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs," *J. of Parallel and Distributed Computing*, vol. 63, no. 11, 2003.

[12] M. Stillwell, F. Vivien, and H. Casanova, "Dynamic fractional resource scheduling for HPC workloads," in *Proc. of the 24th Intl. Parallel and Distributed Processing Symp.*, 2010, to appear.